Noname manuscript No.
(will be inserted by the editor)

An online heuristic selection approach for the Travelling Thief Problem based on Genetic Programming

Mohamed El Yafrani · Marcella Martins · Markus Wagner · Belaïd Ahiod · Myriam Delgado · Ricardo Lüders ·

Received: date / Accepted: date

Abstract In this paper, we investigate the use of hyper-heuristics for the Travelling Thief Problem (TTP). TTP is a multi-component problem, which means it has a composite structure. The problem is a combination between the Travelling Salesman Problem (TSP) and the Knapsack Problem (KP). Many heuristics were proposed to deal with the two components of the problem separately. In this work, we investigate the use of automatic online heuristic selection in order to find the best combination of the different known heuristics. In order to achieve this, we propose a genetic programming based hyper-heuristic called HSGP, and compare it to state-of-the-art algorithms. The experimental results show that the approach is competitive with those algorithms on small TTP instances.

Keywords Heuristic selection · Genetic Programming · Travelling Thief Problem · Multi-component problems

Mohamed El Yafrani (✉) · Belaïd Ahiod
LRIT, associated unit to CNRST (URAC 29)
Mohammed V University in Rabat
B.P. 1014 Rabat, Morocco
E-mail: m.elyafrani@gmail.com, ahiod@fsr.ac.ma

Markus Wagner
Optimisation and Logistics
School of Computer Science
University of Adelaide, Australia
E-mail: markus.wagner@adelaide.edu.au

Marcella S. R. Martins · Myriam R. B. S. Delgado · Ricardo Lüders
Federal University of Technology - Paraná (UTFPR)
Av. Sete de Setembro, 3165. Curitiba PR, Brazil
E-mail: {marcella,myriamdelg,luders}@utfpr.com.br

1 Introduction

Many applications and real-world problems involve combinatorial optimization. These problems are often more complex to solve, once difficulties arise from the extremely large and/or heavily constrained search spaces, and the noisy/dynamic nature of many real-world scenarios [5].

Over the years, a large variety of heuristic methods has been proposed to deal with them. These methods solve problems by discovering a solution in the set of all possible solutions (search space). Non-deterministic search methods such as evolutionary algorithms, local search, and others search algorithms offer an alternative approach to exhaustive search to achieve good solutions for difficult computational problems in a reasonable amount of time. These methods can generate good quality solutions, but there is no guarantee that an optimal solution will be produced.

It is important to highlight that these approaches still find difficulties in terms of adaptation to newly encountered problems, or even new instances of a similar problem. Additionally, these search strategies can be resource-intensive to implement and develop.

Hyper-heuristics aim to address some of these issues. In fact, hyper-heuristics are a trending class of high-level search techniques designed to automate the heuristic design process and raise the level of generality at which search methods operate [34]. Hyper-heuristics are search methodologies for selecting heuristics or generating new heuristics combining and adapting components of heuristics, in order to solve a range of optimization problems. They operate on a search space of heuristics unlike traditional computational search methods which operate directly on a search space of solutions [13].

Genetic Programming (GP) [22] is one of the most commonly used approaches in the field of hyper-heuristics [5]. GP is an evolutionary computation method aiming to evolve a population of programs, traditionally represented as tree structures.

GP has been usually used as a heuristic generation approach. However, in this work we explore the use of GP from a heuristic selection perspective. Heuristic selection using GP has been previously investigated by Nguyen et al. [31] in his adaptive model called GPAM. Hunt et al. [19] proposed another heuristic selection approach for the feature selection problem.

Our proposed hyper-heuristic based on genetic programming is evaluated on the Travelling Thief Problem (TTP), a relatively new NP-hard problem [2]. TTP is an optimization problem that provides interdependent sub-problems, which is a problem aspect often encountered in real-world applications.

Despite the fact that many heuristics were proposed to deal with the TTP components separately, most existing algorithms are based on local search heuristics and meta-heuristic adaptation. None of the proposed approaches investigate the use of the heuristic selection in order to find the best combination of the different heuristics, which is the main motivation of this paper.

Indeed, we believe that the composite structure of the problem makes it a good benchmark for heuristic selection, or more precisely sub-heuristic se-

lection. Therefore, in this paper we propose a heuristic selection framework based on genetic programming. The approach uses well known sub-heuristics in order to evolve combinations of these heuristics in order to find a good model for the instance at hand.

Through this work, we aim especially to answer to the following question: how efficient a genetic programming heuristic selection approach can be to find good quality solutions for the TTP? In order to answer it, we conduct an in-depth experimental study on a subset of the TTP library.

This paper is organized as follows. Section 2 presents the basic concepts about GP and some related works in combinatorial problems using hyper-heuristics. The TTP problem and a brief history of TTP algorithms are introduced in Section 3. Section 4 describes our proposed approach. Our experiments are described in Section 5 and the results are discussed in Section 6. Finally, conclusions and future directions are presented in Section 7.

2 On the use of hyper-heuristics in combinatorial optimization

This section provides a background and some related works in combinatorial optimization problems using hyper-heuristics. Two classes of hyper-heuristics are presented, and the use of genetic programming as a hyper-heuristic is briefly revisited.

2.1 Heuristic selection vs heuristic generation

A hyper-heuristic can be defined as a high-level heuristic (HLH) that controls low-level heuristics (LLH) [11]. Two main categories of hyper-heuristics can be distinguished according to Burke et al. [6]: heuristic selection methodologies and heuristic generation methodologies. Heuristic selection frameworks select a LLH to apply at a given point during the search process. The framework is provided with a set of pre-existing, generally widely known heuristics for solving the target problem. On the other hand, the objective of heuristic generation methodologies is to automatically design new heuristics using components of previously known heuristics [38]. Heuristic generation approaches mostly use a training phase, in which the model is evolved using a subset of problem instances, called the training set. These hyper-heuristics are classified as offline learning hyper-heuristics.

Burke et al. [6] also define other categories related to the nature of the LLH heuristics used in the hyper-heuristic framework. In either case, the set of LLHs being selected or generated can be further split to distinguish between those which construct solutions from scratch (constructive) and those which modify an existing solution (perturbative) [13]. Furthermore, most of the hyper-heuristics approaches incorporate a learning mechanism to assist the selection of LLH during the solution process. Hyper-heuristics which apply online learning continuously adapt throughout the search process based on the

feedback they receive. Hyper-heuristics using offline learning train the model on a subset of instances before being applied to another, frequently larger, set of unseen instances.

Some related works present good results for automating the design and adaption of heuristic methods with hyper-heuristics. Many meta-heuristics and machine learning techniques have been used in the context of heuristic selection.

Previous works describe hyper-heuristic frameworks that rank adaptively the LLHs based on a select function [9, 10]. The select function is the weighted sum of heuristic performance, joint performance of pairs of heuristics, and CPU time from the previous time the LLH was called. So the ranks are used to decide which heuristic to select in the next call. This method is simple and can be easily applied to new problem domains.

Another interesting method was developed by Krasnogor and Smith [24]. The approach shows how a simple inheritance mechanism is capable of learning what is the best local search heuristic to use at different stages of the search process. An individual is composed of its genetic material and its memetic material. The memetic material specifies the strategy the individual will use to apply local search in the vicinity of the solution encoded in its genetic part. This method was applied to solve different combinatorial optimization problems and showed very promising results.

Local search methods have also been used for heuristic selection such as the Tabu Search based hyper-heuristic [7]. The authors proposed a hyper-heuristic framework for timetabling and rostering in which heuristics compete using rules based on the principles of reinforcement learning. A tabu list is maintained in order to prevents certain heuristics from being chosen at certain times during the search. This framework was applied in two different problem domains and obtained good results on both.

In [35], the authors used a genetic algorithm as a hyper-heuristic that learns heuristic combinations for solving one-dimensional bin-packing problems. Another approach described in [11] proposed a very basic ACO based hyper-heuristic to evolve sequences of (at most) five heuristics for 2D packing. Both approaches provide competitive results.

In [12], the authors proposed a Simulated Annealing hyper-heuristic for the shipper rationalization problem, and the results showed that the approach was able to incorporate changes without the need for extensive experimentation to determine their impact on solution speed or quality.

2.2 Genetic Programming as a Hyper-heuristic

In recent years, Genetic Programming (GP) has been widely applied as a hyper-heuristic framework due its flexible structure, that can be used to build either construction or perturbation heuristics, with successful implementations in real world problems [20, 21].

GP [22, 23] is an evolutionary computation method that can be classified as a machine learning technique. GP is used to evolve populations of computer programs traditionally represented as tree structures. Each node in the tree returns a value to its parent node. The internal nodes have one or more children nodes, and they return a value obtained by performing operations on the values returned by their children nodes. The tree's internal nodes are referred to as functions, and leaf nodes are referred to as terminals.

Besides that, GP has a set of parameters that must be set, such as the size of the population, and the stopping criterion. The setting also includes the specification of the functions and terminals, along with the fitness function, which ultimately drives the evolutionary process [5].

The terminal set is the mechanism through which the problem state is made available to the program, changing their value as the problem state changes. The functions of a GP tree manipulate the values supplied to the program by the terminals, and as such their defining feature is that they take one or more arguments, which can be the values returned by terminal nodes or other function nodes. Each individual computer program in the population is measured in terms of how well it performs in the particular problem environment [25]. Genetic operators such as crossover and mutation are performed on the individuals at each generation. One of the advantages highlighted is GP relies on expert knowledge to define its terminal and function sets [6].

One of the most common applications of GP as a hyper-heuristic is the automatic generation of heuristics. For example, Bölte and Thonemann [1] successfully adapted GP to learn new heuristics. The proposed method used standard GP to evolve annealing schedule functions in simulated annealing to solve the Quadratic Assignment Problem (QAP). Another example is the use of GP as a hyper-heuristic methodology to generate constructive heuristics in order to solve the Multidimensional 0-1 Knapsack Problem [13]. The results over a set of standard benchmarks showed that this approach leads to human competitive results. Furthermore, the work in [4] presented a GP system which automated the heuristic generation process producing human-competitive heuristics for the online bin packing problem. The framework evolved a control program that rates the suitability of each bin for the next piece, using the attributes of the pieces to be packed and the bins.

The authors in [27] also proposed a heuristic generation evolution with GP for the TTP. The approach consists in employing GP to evolve a gain and a picking function, respectively, in order to replace the original manually designed item selection heuristic in TSMA (Two-Stage Memetic Algorithm). The aim of this paper was to conduct a more systematic investigation on the item picking heuristic in TSMA.

According to Nguyen et al. [31], GP can be also applied for heuristic selection methods for NP-hard combinatorial optimization problems. The author in [18] proposed the Composite heuristic Learning Algorithm for SAT Search (CLASS). CLASS is a GP framework that discovers satisfiability testing (SAT) local search heuristics. The evolved heuristics were shown to be competitive compared to well-known SAT local search algorithms.

Nguyen et al. [31] investigated a GP based hyper-heuristic approach that evolves adaptive mechanisms called GPAM. The hyper-heuristic chooses from a set of LLH and constructs an adaptive mechanism, which is evolved simultaneously with the problem solution, providing an online learning system. Results showed that GPAM presented good quality solutions that performed competitively alongside existing hyper-heuristics for the MAX-SAT, bin packing and flow-shop scheduling problem domains.

Another GP-based heuristic selection framework was proposed in [19] for the feature selection problem. The proposed method evolves new heuristics using some basic components (building blocks). The evolved heuristics act as new search algorithms that can search the space of subsets of features.

A recent work presented in [8] introduced a hyper-heuristic with a GP search process that includes a self-tuning technique to dynamically update the crossover and mutation probabilities during a run of GP. The approach assigns different crossover and mutation probabilities to each candidate solution. In order to evaluate the performance of the proposed approach, the authors considered seven different symbolic regression test problems.

Our proposal is based on LLH selection methodology as presented in [19], but in our case, the framework can learn from feedback concerning heuristic performance throughout the search process, similar what was done in [31]. This approach is tested over TTP problem instances. TTP instances were also investigated by Mei et al. [27] from a hyper-heuristic perspective. However, the main difference between the work proposed in [27] and our approach is that we aim to investigate the LLHs combination instead of evolving the item selection function.

3 The Travelling Thief Problem (TTP)

The Travelling Thief Problem (TTP) is an artificially designed problem that combines the Travelling Salesman Problem and the Knapsack Problem. The motivation of designing such a problem is to provide a benchmark model closer to real optimization problems, which in many situations, are composed of multiples interacting sub-problems.

In this section, the TTP is presented and mathematically defined. Then, some of best performing algorithms are briefly introduced.

3.1 TTP and hyper-heuristics

The Travelling Thief Problem was first introduced by Bonyadi et al. [2] with the purpose of providing a benchmark for problems with multiple interdependent components. The problem was then simplified and reformulated in [32] and many benchmark instances were proposed in the same paper.

Since the problem has two components, many approaches focus on designing a heuristic for each part of the problem. We believe this presents a good

testbed for heuristic selection in particular and hyper-heuristics in general. Indeed, many local search routines and disruptive operators have been proposed for the two components of the problem. Therefore, a heuristic selection approach would provide a way to automatically combine heuristics and mutation operators, in order to determine the sequence that they must be applied to assure the best achievable objective value.

A first attempt on using hyper-heuristics to solve the problem was proposed by Mei et al. [27] in an above mentioned paper. The approach consists on using genetic programming to evolve packing routines for the KP component of the problem.

A recent paper by [41] presents a detailed study on algorithm selection for TTP. The paper uses 21 algorithms for the TTP and all 9720 instances in order to create a portfolio of algorithms. This portfolio is used to determine what algorithm performs best for what instance.

3.2 Problem Definition

In the TTP, we are given a set of n cities, the associated matrix of distances d_{ij} between cities i and j, and a set of m items distributed among the n cities. Each item k is characterized by its profit p_k and weight w_k. A thief must visit all cities exactly once, stealing some items on the road, and return to the first city.

The total weight of the collected items must not exceed a specified capacity W. In addition, we consider a renting rate per time unit R that the thief must pay at the end of the travel, and the maximum and minimum velocities denoted v_{max} and v_{min} respectively. Each item is available in only one city, and we note $A_i \in 1, \ldots, n$ the availability vector. A_i contains the reference to the city that contains the item i.

Naturally, a TTP solution is coded in two parts. The first is the tour $x = (x_1, \ldots, x_n)$, a vector containing the ordered list of cities. The second is the picking plan $z = (z_1, \ldots, z_m)$, a binary vector representing the states of items (0 for packed, and 1 for unpacked).

To make the sub-problems mutually dependent, the speed of the thief changes according to the knapsack weight. Therefore, the thief's velocity at city x is defined in Equation 1.

$$v_x = v_{max} - C \times w_x \tag{1}$$

where $C = (v_{max} - v_{min})/W$ is a constant value, and w_x the weight of the knapsack at city x.

We note $g(z)$ the total value of all collected items and $f(x, z)$ the total travel time which are defined in Equations 2 and 3 respectively.

$$g(z) = \sum_{m} p_m \times z_m \tag{2}$$
$$\text{subject to} \sum_{m} w_m \times z_m \leq W$$

$$f(x, z) = \sum_{i=1}^{n-1} t_{x_i, x_{i+1}} + t_{x_n, x_1} \tag{3}$$

The objective is to maximize the travel gain, as defined in Equation 4, by finding the best tour and picking plan.

$$G(x, z) = g(z) - R \times f(x, z) \tag{4}$$

Note that it has been shown in [2, 30] that optimizing the sub-problems in isolation, even to optimality, does not guarantee finding good solutions to the overall problem.

3.3 A brief history of TTP algorithms

Since the appearance of the TTP, several heuristic algorithms were proposed to solve it. The first version of the problem was proposed by Bonyadi et al. [2], in which an item can appear in multiple cities. Mei et al. [30] investigated the interdependence in this first formulation of the TTP, and proposed two algorithms to solve it. The algorithms are a Cooperative Coevolution heuristic and Memetic Algorithm.

Afterwards, [32] proposed a simplified version of the problem where an item can occur only once in a city. The paper also presented a very large library for the TTP containing 9720 instances, and three simple heuristics to solve these instances. The heuristics are a constructive heuristic named Simple Heuristic (SH), a Random Local Search (RLS), and a (1+1) Evolutionary Algorithm (EA). EA and RLS were able to obtain a positive gain, but were lately surpassed by more sophisticated heuristics.

Mei et al. [28] introduced a memetic algorithm named MATLS able to solve very large instances. The algorithm's success is mainly due to the use of domain knowledge to speed up local search, and the efficient greedy packing routines. The algorithm is shown to be efficient for instances having more than 10000 cities. The same framework was later used to design a GP-based heuristic generation approach in order to improve the packing process [29].

The paper by Bonyadi et al. [3] also presented a framework called CoSolver. The idea is to handle the components of the TTP separately, but maintain a communication tunnel between the two sub-problems in order to take interdependence into consideration.

Faulkner et al. [17] implemented multiple local search and greedy algorithms. The algorithms are combined in different fashions in order to design

more sophisticated approaches. The authors proposed two families of heuristics as a result of the combinations, simple heuristics (S1-S5) and complex ones (C1-C6). Surprisingly, the best performing heuristic is S5 which belongs to simple heuristics group.

Most proposed heuristics use a Lin-Kernighan tour to initialize the solution. In a tentative to investigated this bias in heuristic design for the TTP, Wagner [40] investigated longer tours using the Max-Min Ant System. This approach focuses on improving the tour accordingly to the overall TTP problem instead of using a Lin-Kernighan tour, which is designed for the TSP component independently. The approach is shown to be very efficient for small TTP instances.

Recently, El Yafrani and Ahiod [16] presented two heuristics. The first is a memetic algorithm using 2-opt and bit-flip local search heuristics, named MA2B. The second is a combination of a 2-opt local search and a simulated annealing based heuristic for efficient packing, named CS2SA. The two proposed heuristics were shown to be very competitive to other heuristics such as MATLS and S5. MA2B particularly performs well on small instances, while CS2SA was more efficient on large instances.

Wu et al. [43] investigated the impact of the renting rate on a special case of the TTP in which the tour is supposed fixed called the Nonlinear Knapsack Problem. The paper presents an in-depth theoretical study of the renting rate and its effect on the hardness of a given instance. The authors also proposed an approach to generate hard instances based on theoretical and experimental study.

Finally, the authors in [15] focus on designing a TTP specific neighborhood instead of using a sequential structure as in most heuristics. The paper presented the use of speedups in the context of the proposed neighborhoods. The results show that this approach was competitive to EA and RLS on different small instances. However, the algorithms lack of exploration capabilities as they are mainly designed for exploitation purposes.

4 The proposed approach

In our proposed approach, the goal is to apply GP as a heuristic selection technique, aiming to evolve combinations of heuristics in order to find good problem solutions. In this section we explain how this strategy can be implemented detailing the proposed algorithm named HSGP (which stands for Heuristic Selection based on Genetic Programming).

The motivation behind choosing GP in the context of heuristic selection is mainly due to two properties. Firstly, GP trees are flexible, which means that evolving a GP tree is easier than using a fixed size array. Secondly, GP preserves the correlation between the terminals in sub-trees. The correlations are transferred to the offspring to produce new trees using mainly crossover.

In this section, the details of our GP adaptation are presented.

4.1 Representation

In the GP population, every individual is encoded as a tree that represents the program to be executed. The tree's internal nodes are functions, while leaf nodes are terminals, or LLHs.

The functions set we use is composed by connectors, which are used to sequentially execute the child sub-tree from left to right. This is rather a simple adaptation compared to other possibilities like using conditional statements and loops. Nevertheless, these connectors allow representing different combinations of LLHs heuristics. Additionally, it is possible to use multiple connectors with different numbers of child sub-trees.

In our implementation, we only use one type of connectors, which we refer to as *prog_2*. The *prog_2* connector executes the sub-tree from left to right sequentially.

On the other hand, we use a set of several terminals. The terminals are either a component heuristic or a disruptive operation. In the following, we present a list of the used terminals:

- **term_kpbf:** A neighborhood search heuristic targeting the KP part. This heuristic uses a simple bit-flip local search empowered with speedup techniques. It is part of the memetic algorithm MATLS proposed in [28].
- **term_kpsa:** A simulated annealing adapted for the KP sub-problem. It is used in CS2SA presented in [16].
- **term_tsp2opt:** A 2-opt based local search heuristic used for the TSP component. It is used in many TTP algorithms [14, 16, 29, 30].
- **term_otspswap:** A disruptive move for the TSP sub-problem that randomly swaps two cities.
- **term_otsp4opt:** A double bridge move for the TSP sub-problem that randomly selects the cities.
- **term_okpbf20:** A disruptive routine that toggles the state of 20% of the picking plan items.
- **term_okpbf30:** A disruptive routine that toggles the state of 30% of the picking plan items.
- **term_okpbf40:** A disruptive routine that toggles the state of 40% of the picking plan items.

The last three disruptive terminals were provided in order to enlarge the search space.

An example of a tree individual can be seen in Figure 1.

4.2 The fitness function

The fitness of a GP individual $fitness_{GP}$ depends on its performance on the given instance. First, the tree is parsed in-order to get the list of leave nodes. These are then applied sequentially on the problem instance, starting from an initial TTP solution. The fitness of the individual is set to the achieved TTP objective when the processing of the list of leave nodes is completed.

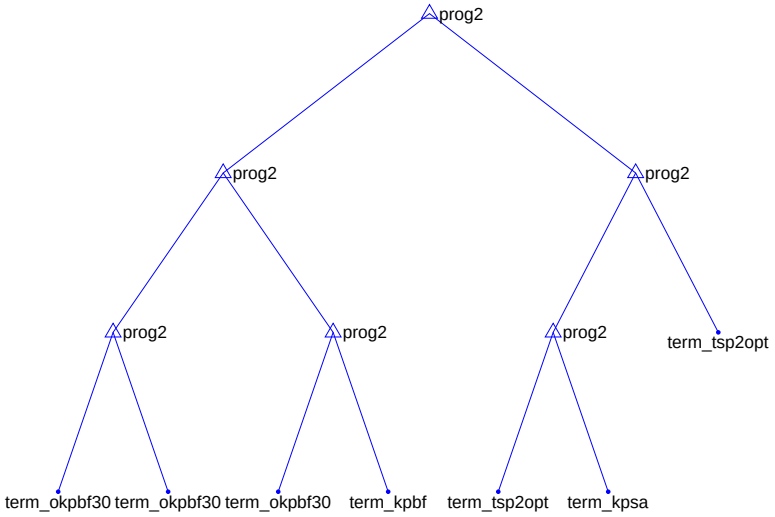

Fig. 1: Example of a GP individual. The terminals are TTP heuristics, while the internal nodes represent connectors.

4.3 The framework

The main steps performed by HSGP are described in Algorithm 1. The proposed algorithm makes use of specific strategies presented in the next subsections.

4.3.1 Initialization

The Initialization process loads the problem instance, sets the GP parameters GP_{param} and generates an initial population of random compositions of functions and terminals of the problem, Steps 1, 2 and 3, respectively.

There are different approaches to generate the random initial population [25]. In this work, we use the *Full Initialization* method [22] in order to provide full trees and to increase the initial search space.

In this method, the initial individuals are generated according to a given maximum depth. Nodes are taken at random from the function set until the maximum tree depth is reached, and the last depth level is limited to terminal nodes. As a result, trees initialized with this method will be balanced with the same length for all individuals of the initial population. This method prevents small shapes to be generated, and enlarges the genetic operations possibilities.

Algorithm 1 HSGP framework

INPUT: $Instance$: problem instance
GP_{param}: set of GP parameters
N: population size
$Max_{runtime}$: maximum runtime
Max_{ger}: maximum number of generations
OUTPUT: T: the final individual tree
{Initialization}
1: $\mathbf{I} \leftarrow$ LoadInstance($Instance$)
2: $\mathbf{P} \leftarrow$ SetParams(GP_{param})
3: $Pop^1 \leftarrow$ RandomGenerate(N, I, P) {initial population}
4: $g \leftarrow 1$;
{GP: main loop}
5: **while** $g \leq Max_{ger}$ and $\neg Max_{runtime}$ **do**
6: Pop^g.**fitness** $\leftarrow$EvaluateFitness(Pop^g, I, P)
7: $Pop_{\text{offspring}} \leftarrow$GeneticOperator($Pop^g, P$); {offspring population}
8: $Pop_{\text{offspring}}$.**fitness** $\leftarrow$EvaluateFitness($Pop_{\text{offspring}}, I, P$)
{GP: Survival}
9: $Pop^{g+1} \leftarrow$ Survival($\{Pop^g \cup Pop_{\text{offspring}}\}, N$);{new population}
10: $g \leftarrow g + 1$
11: **end while**
{GP: best individual}
12: $T_{best} \leftarrow$ SelectBest(Pop^{g-1})

4.3.2 GP: main loop

The proposed GP algorithm searches the space of possible heuristic combinations. In the context of our implementation, each GP program is a tree of heuristics. The program is executed in an infix manner to search the space of possible problem solutions in order to find good ones.

The *EvaluateFitness*, in Step 6, calculates $fitness_{GP}$ based on the TTP objective function. In Step 7 *GeneticOperator* applies genetically inspired operations of mutation and crossover in selected parents individuals, in order to produce a new population of programs $Pop_{\text{offspring}}$. The selection of these parents individuals is probabilistically based on fitness. That is, better individuals are more likely to have more child programs than inferior individuals.

The most commonly employed method for selecting these parents in GP is tournament selection [22], which is used in our framework. This method selects a random number of individuals from the population and the best of them is chosen[1].

The genetic operators adopted for the selected parents are standard tree crossover and standard mutation. Given two parents, standard crossover randomly selects a crossover point in each parent tree. Then, it creates the offspring by replacing the sub-tree rooted at the crossover point in a copy of the first parent with a copy of the sub-tree rooted at the crossover point in the second parent [25]. In standard tree mutation, a randomly created new tree

[1] We apply the Lexicographic Parsimony Pressure technique [26]. If two individuals are equally fit, the tree with less nodes is chosen as the best. This technique has shown to effectively control bloat in different types of problems [37].

replaces a randomly chosen branch (excluding the root node) of the parent tree [37].

This offspring population of programs $Pop_{\text{offspring}}$ is evaluated using fitness function and merged with the current population Pop^g, where g is the generation number. However, N individuals are selected in *Survival* process to continue in the evolutionary process as a new population Pop^{g+1}, as can be seen in Step 9. The best individual from both Pop^g and $Pop_{\text{offspring}}$ is kept in the new population Pop^{g+1}. The fittest individuals from $Pop_{\text{offspring}}$ followed by the fittest ones from Pop^g compose the remaining $N-1$ individuals from Pop^{g+1}.

This process is iteratively performed until the termination criterion (Max_{ger} or $Max_{runtime}$) has been satisfied, whichever comes first. At the end the best individual T_{best} is selected from the population, according *SelectBest* function in Step 12.

5 Experiments

5.1 Benchmark Instances

The experiments conducted in this paper are performed on a comprehensive subset of the TTP benchmark instances [2] from [32]. The characteristics of these instances vary widely, and in this work we consider the following diversification parameters:

- The number of cities is based on TSP instances from the TSPlib, described in [33]
- For each TSP instance, there are three different types (we will refer to this parameter as T) of knapsack problems: uncorrelated (unc), uncorrelated with similar weights (usw), and bounded strongly correlated (bsc) types
- For each TSP and KP combination, the number of items per city (item factor, denoted F) is $F \in \{1, 5, 10\}$
- For each TTP configuration, we use 3 different knapsack capacities $C \in \{1, 5, 10\}$. C represents a capacity class.

To evaluate the proposed hyper-heuristic, we use four representative small sized TTP instance groups: *eil51*, *berlin52*, *eil76*, and *kroA100*. Therefore, using this setting, a total of 108 instances are considered in this paper.

5.2 GP tuning

The hyper-heuristic framework is developed based on GPLAB [36], a GP toolbox for MATLAB. The GP parameters we used for the experiments are shown in Table 1.

[2] All TTP instances can be found in the website: http://cs.adelaide.edu.au/ optlog/research/ttp.php.

Table 1: Parameters of HSGP algorithm.

Description	Value
Population size N	10
Maximum runtime $Max_{runtime}$	$600s$
Maximum number of generations Max_{ger}	100
Tournament size	5
Crossover Rate	0.9
Mutation Rate	0.1
Reproduction Rate	0.1
Max Depth	6
Min Depth	2
Tournament size	2

Additionally, we use the *Full Initialization* technique alongside a minimum depth threshold of 2 in order to encourage the use of deeper initial trees. The reproduction (replication) rate is 0.1, meaning that each selected parent has a 10% chance of being copied to the next generation without suffering the action of the genetic operators. Standard tree mutation and standard crossover (with uniform selection of crossover and mutation points among different tree levels) were used with probabilities of 0.1 and 0.9, respectively. Also, a maximum tree depth *Max Depth=6* is imposed to any new branch created in order to avoid bloat.

All algorithms are performed on a core i3-2370M CPU 2.40GHz machine with 4 GB of RAM, running Linux, for a maximum of 10 minutes per instance. In addition, 10 independent executions are conducted for each algorithm. The obtained results are then used to perform a statistical test in order to compare the performance of HSGP to the other algorithms.

6 Results and Discussion

Comparing different optimization techniques experimentally involves the notion of performance. This section presents a comparison between our HSGP approach and three other algorithms for the TTP.

To measure the quality of the algorithms, we consider for each algorithm the average objective score of 10 independent runs, and we consider the best objective score found by any algorithm. The ratio between the average and the best objective values found gives us the approximation ratio. According to Wagner [40], this ratio allows us to compare the performances across the chosen set of instances, since the objective values vary across several orders of magnitude.

In Figure 2, we show a summary of over 432 (108 instances and 4 algorithms) average approximation ratios as trend lines. The curves are polynomials of degree six, as considered in [40].

We can observe in Figure 2 that our HSGP approach outperforms both CS2SA and S5 for almost all TTP instances. For the instances with 100 cities,

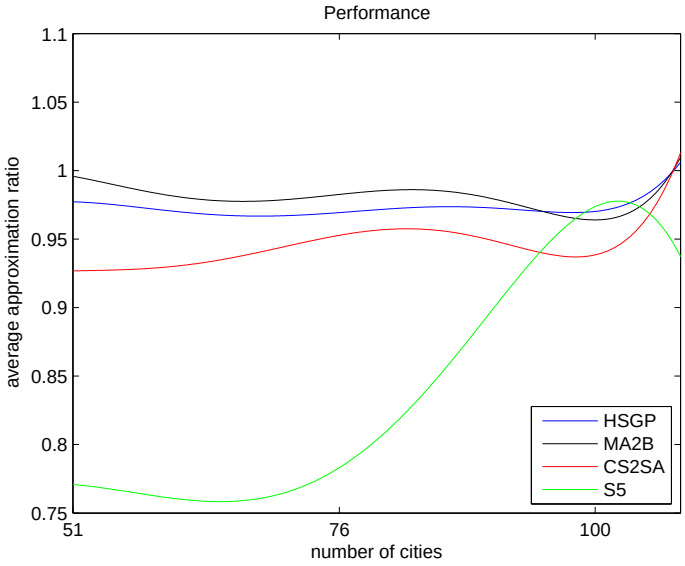

Fig. 2: Performance comparison - Summary of results shown as trend lines. The $x-$axis represents the 108 instances: 27 of *eil51*; 27 of *berlin52*; 27 of *eil76*; and 27 of *kroA100*.

HSGP is competitive with MA2B and has higher average approximation ratio. However, for most instances, MA2B still beats all the other approaches.

A deeper analysis can show that individual approaches do not strictly follow the trends. Figure 3 in Appendix A presents the average approximation ratios achieved in 10 independent runs.

In order to provide a statistical analysis of the results, the A-test (Vargha-Delaney A measure [39]) was applied. This test tells us how often, on average, one technique outperforms the other. Its a non-parametric test called the measure of stochastic superiority.

The hypothesis of stochastic equality ($A = 0.5$) can be tested by using any of several methods, and the most related is the Mann-Whitney-Wilcoxon rank sum test [42], which was used and has been executed with a confidence level of 95% ($\alpha = 5\%$) .

The A-test test returns a value between 0 and 1, representing the probability that a randomly selected observation from one sample is bigger than a randomly selected observation from other sample. Therefore it provides how much the two samples overlap. The two samples are composed by objective values from each algorithm run. Then each sample has 10 runs.

Tables 2, 3, 4 and 5 show the statistical analysis of a pairwise comparison between these algorithms for each instance, respectively.

When the p-value is exactly 0.5, there is no statistical difference between the two techniques. When the p-value is less than 0.5, the first technique

has the worse performance. Lastly, when the p-values is greater than 0.5, the second technique is the worst performing one.

The entries representing when HSGP is showing a better performance than another approach are emphasized (bold).

Table 2: p-values for statistical A-test with a significance level of $\alpha = 5\%$ over *eil51* instances.

Algorithm	HSGP x ma2b	HSGP x cs2sa	HSGP x s5
$eil51(01, bsc, 01)$	0.0001	**1.0000**	**1.0000**
$eil51(05, bsc, 01)$	0.0001	0.33	**1.0000**
$eil51(10, bsc, 01)$	0.0001	0.23	**1.0000**
$eil51(01, bsc, 05)$	0.11	**1.0000**	**1.0000**
$eil51(05, bsc, 05)$	0.3	**0.6500**	**1.0000**
$eil51(10, bsc, 05)$	0.1	0.5	**1.0000**
$eil51(01, bsc, 10)$	**0.7300**	**1.0000**	**1.0000**
$eil51(05, bsc, 10)$	**0.6500**	**0.6500**	**1.0000**
$eil51(10, bsc, 10)$	0.055	**0.5500**	**1.0000**
$eil51(01, usw, 01)$	0.3	**1.0000**	0.0001
$eil51(05, usw, 01)$	0.15	**0.7700**	**1.0000**
$eil51(10, usw, 01)$	0.07	**1.0000**	0.0001
$eil51(01, usw, 05)$	0.0001	**0.6000**	0.0001
$eil51(05, usw, 05)$	0.0001	**0.6000**	**1.0000**
$eil51(10, usw, 05)$	0.0001	**0.5500**	**1.0000**
$eil51(01, usw, 10)$	0.06	**1.0000**	0.0001
$eil51(05, usw, 10)$	0.0001	**0.5000**	**1.0000**
$eil51(10, usw, 10)$	0.0001	**1.0000**	**1.0000**
$eil51(01, unc, 01)$	0.0001	**1.0000**	0.0001
$eil51(05, unc, 01)$	0.09	**1.0000**	**1.0000**
$eil51(10, unc, 01)$	0.0001	**0.9700**	0.0001
$eil51(01, unc, 05)$	0.06	**0.8500**	0.0001
$eil51(05, unc, 05)$	0.45	0.5	**1.0000**
$eil51(10, unc, 05)$	0.0001	**1.0000**	**1.0000**
$eil51(01, unc, 10)$	0.0001	**1.0000**	**1.0000**
$eil51(05, unc, 10)$	0.4	0.5	**1.0000**
$eil51(10, unc, 10)$	0.0001	0.5	**1.0000**

We can observe in Table 2 that MA2B shows the best results for *eil51* instances. However, HSGP is better than CS2SA and S5 for almost all instances. For some instances there is no statistical difference between HSGP and CS2SA. Such instances include $eil51(10, bsc, 05)$, $eil51(05, unc, 05)$, $eil51(05, unc, 10)$, and $eil51(10, unc, 10)$.

In Table 3 HSGP is better than MA2B for some instances with a large number of items, and when the items weight and profit are uncorrelated or uncorrelated with similar weights. This can be clearly observed in the following instances: $berlin52(10, usw, 05)$, $berlin52(10, usw, 10)$, $berlin52(10, unc, 01)$, $berlin52(10, unc, 05)$, and $berlin52(10, unc, 10)$. We can also note that HSGP is better than S5 and CS2SA for almost all *berlin52* instances, except for some instances with bounded strongly correlated profit/weight.

We can note in Table 4 that HSGP is better than S5 for all *eil76* instances, and its better than CS2SA for most instances. Additionally, HSGP was able to outperform MA2B for some instances.

Finally, the p-values reported in Table 5 show that HSGP is better than S5 for all *kroA100* instances and better or at least statistically similar to CS2SA.

Table 3: p-values for statistical A-test with a significance level of $\alpha = 5\%$ over *berlin52* instances.

Algorithm	HSGP x ma2b	HSGP x cs2sa	HSGP x s5
$berlin52(01, bsc, 01)$	0.01	**0.9400**	0.2
$berlin52(05, bsc, 01)$	0.0001	**0.7100**	0.0001
$berlin52(10, bsc, 01)$	0.0001	0.11	0.0001
$berlin52(01, bsc, 05)$	0.0001	0.2	**1.0000**
$berlin52(05, bsc, 05)$	0.2	**0.7200**	**1.0000**
$berlin52(10, bsc, 05)$	0.09	**0.9100**	**1.0000**
$berlin52(01, bsc, 10)$	0.015	**1.0000**	**1.0000**
$berlin52(05, bsc, 10)$	0.48	**1.0000**	**1.0000**
$berlin52(10, bsc, 10)$	0.39	**1.0000**	**1.0000**
$berlin52(01, usw, 01)$	0.385	**0.9550**	**1.0000**
$berlin52(05, usw, 01)$	0.0001	**1.0000**	**1.0000**
$berlin52(10, usw, 01)$	0.0001	0.09	**1.0000**
$berlin52(01, usw, 05)$	**0.5700**	**1.0000**	**1.0000**
$berlin52(05, usw, 05)$	0.27	**0.9000**	**1.0000**
$berlin52(10, usw, 05)$	**0.5300**	**1.0000**	**1.0000**
$berlin52(01, usw, 10)$	0.25	**1.0000**	**1.0000**
$berlin52(05, usw, 10)$	0.17	**1.0000**	**1.0000**
$berlin52(10, usw, 10)$	**0.5900**	**1.0000**	**1.0000**
$berlin52(01, unc, 01)$	0.29	**1.0000**	**1.0000**
$berlin52(05, unc, 01)$	0.26	**1.0000**	**1.0000**
$berlin52(10, unc, 01)$	**0.5400**	**1.0000**	**1.0000**
$berlin52(01, unc, 05)$	0.015	**1.0000**	**1.0000**
$berlin52(05, unc, 05)$	0.08	**0.9000**	**1.0000**
$berlin52(10, unc, 05)$	**0.6600**	**1.0000**	**1.0000**
$berlin52(01, unc, 10)$	0.25	**1.0000**	**1.0000**
$berlin52(05, unc, 10)$	0.1	**1.0000**	**1.0000**
$berlin52(10, unc, 10)$	**0.5200**	**1.0000**	**1.0000**

Table 4: p-values for statistical A-test with a significance level of $\alpha = 5\%$ over *eil76* instances.

Algorithm	HSGP x ma2b	HSGP x cs2sa	HSGP x s5
$eil76(01, bsc, 01)$	0.08	**0.9200**	**1.0000**
$eil76(05, bsc, 01)$	0.0001	0.25	**1.0000**
$eil76(10, bsc, 01)$	0.0001	0.355	**1.0000**
$eil76(01, bsc, 05)$	0.06	**1.0000**	**1.0000**
$eil76(05, bsc, 05)$	0.02	**1.0000**	**1.0000**
$eil76(10, bsc, 05)$	**0.6000**	0.465	**1.0000**
$eil76(01, bsc, 10)$	0.26	0.4	**1.0000**
$eil76(05, bsc, 10)$	0.04	**0.8000**	**1.0000**
$eil76(10, bsc, 10)$	0.0001	**0.7000**	**1.0000**
$eil76(01, usw, 01)$	0.19	**1.0000**	**1.0000**
$eil76(05, usw, 01)$	**0.7000**	0.35	**1.0000**
$eil76(10, usw, 01)$	0.15	0.11	**1.0000**
$eil76(01, usw, 05)$	0.2	**1.0000**	**1.0000**
$eil76(05, usw, 05)$	0.48	**0.9500**	**1.0000**
$eil76(10, usw, 05)$	**0.5200**	**0.8000**	**1.0000**
$eil76(01, usw, 10)$	0.03	**0.6000**	**1.0000**
$eil76(05, usw, 10)$	0.0001	**1.0000**	**1.0000**
$eil76(10, usw, 10)$	0.05	0.1	**1.0000**
$eil76(01, unc, 01)$	0.0001	**1.0000**	**1.0000**
$eil76(05, unc, 01)$	0.395	**1.0000**	**1.0000**
$eil76(10, unc, 01)$	0.17	**1.0000**	**1.0000**
$eil76(01, unc, 05)$	0.0001	**0.7500**	**1.0000**
$eil76(05, unc, 05)$	0.025	**0.9500**	**1.0000**
$eil76(10, unc, 05)$	0.135	**1.0000**	**1.0000**
$eil76(01, unc, 10)$	0.315	**0.9500**	**1.0000**
$eil76(05, unc, 10)$	0.0001	0.1	**1.0000**
$eil76(10, unc, 10)$	0.0001	0.2	**1.0000**

Table 5: p-values for statistical A-test with a significance level of $\alpha = 5\%$ over *kroA100* instances.

Algorithm	HSGP x ma2b	HSGP x cs2sa	HSGP x s5
$kroA100(01, bsc, 01)$	**0.5400**	**0.7950**	**1.0000**
$kroA100(05, bsc, 01)$	0.0001	**0.7500**	**1.0000**
$kroA100(10, bsc, 01)$	0.29	**0.9800**	**0.8000**
$kroA100(01, bsc, 05)$	**0.6200**	**0.8000**	**1.0000**
$kroA100(05, bsc, 05)$	0.185	**1.0000**	**1.0000**
$kroA100(10, bsc, 05)$	0.175	**1.0000**	**1.0000**
$kroA100(01, bsc, 10)$	0.3	**0.8500**	**1.0000**
$kroA100(05, bsc, 10)$	0.1	0.5	**1.0000**
$kroA100(10, bsc, 10)$	0.5	0.5	**1.0000**
$kroA100(01, usw, 01)$	0.355	**0.8000**	**1.0000**
$kroA100(05, usw, 01)$	0.425	**0.9000**	**1.0000**
$kroA100(10, usw, 01)$	**1.0000**	**1.0000**	**1.0000**
$kroA100(01, usw, 05)$	**0.8900**	**1.0000**	**1.0000**
$kroA100(05, usw, 05)$	0.37	**1.0000**	**1.0000**
$kroA100(10, usw, 05)$	**0.6000**	**0.8000**	**1.0000**
$kroA100(01, usw, 10)$	0.11	**1.0000**	**1.0000**
$kroA100(05, usw, 10)$	**1.0000**	0.5	**1.0000**
$kroA100(10, usw, 10)$	0.15	**0.7000**	**1.0000**
$kroA100(01, unc, 01)$	0.0001	**0.65**	**1.0000**
$kroA100(05, unc, 01)$	**0.55**	**0.55**	**1.0000**
$kroA100(10, unc, 01)$	0.0001	0.5	**1.0000**
$kroA100(01, unc, 05)$	**0.78**	**1.0000**	**1.0000**
$kroA100(05, unc, 05)$	0.0001	0.5	**1.0000**
$kroA100(10, unc, 05)$	**0.8**	0.5	**1.0000**
$kroA100(01, unc, 10)$	**0.64**	**1.0000**	**1.0000**
$kroA100(05, unc, 10)$	0.0001	0.5	**1.0000**
$kroA100(10, unc, 10)$	0.0001	0.5	**1.0000**

We can also observe that HSGP presents good results when compared with MA2B. For some instances HSGP is better or at least equal than MA2B.

We believe that the superior performance that MA2B shows is mainly due to two properties. The first is the use of a population of solutions instead of a single solution as is the case with the building blocks of our hyper-heuristic, S5, and CS2SA. The second advantage is the excellent balance between diversification (disruptive crossover and strong mutations) and intensification (fast local search).

Our proposed framework is still a preliminary attempt to investigate the use of heuristic selection for the TTP. We used disruptive operators to increase the search space exploration, which worked to some extent, but needs further improvement.

Nevertheless, even with less sophisticated sub-heuristics, HSGP was able to beat CS2SA which is composed of the same local search terminals used in our model. Although CS2SA is a faster algorithm, the improvement brought by HSGP was significant regarding the solution quality.

Perhaps a fairer comparison would be against S5, due to the fact that it is also very time consuming. Indeed, the reported results show clearly that HSGP outperformed S5 algorithm on the majority of TTP instances.

7 Conclusions and Future Work

In this paper, we studied the Travelling Thief Problem, an NP-hard multi-component problem, from a hyper-heuristic perspective. Firstly, we briefly revisited various hyper-heuristic techniques used in the field of combinatorial optimization. Then, the problem was formally defined and state-of-the-art algorithms were revisited.

The main focus of this work was to investigate the use of heuristic selection for the TTP. Therefore, we proposed a heuristic selection technique based on a GP framework in order to search the best combination between sub-heuristics and disruptive operators.

We have analyzed the performance of the proposed approach on small sized TTP instances, considering an average of approximation ratio. We also provided a statistical analysis of a pairwise comparison between our approach and three other state-of-the-art algorithms.

Based on the experiments we can conclude that for the small instances addressed in this work, the proposed heuristic selection is competitive when compared with the other algorithms investigated. The explanation is based on the fact that GP framework can obtain good combinations of LLH as well as provide a high diversity of the search space. Additionally, GP trees structure have preserved the correlation between the terminals in sub-trees, and this have been transferred to the offspring population. It is interesting to note that no parameter tuning on the HSGP has been performed.

In the future, we can use a self-tuning technique to dynamically update the GP parameters during a run. Furthermore, another interesting research direction is the design of an offline learning GP framework. Such an approach would provide a model evolved on a training set, that is able to solve other unseen instances.

Acknowledgements M.Martins acknowledges CAPES/Brazil. M.Delgado acknowledges CNPq grant Nos.: 309197/2014-7 (Brazil Government).

A Appendix

In this appendix we provide a closer look of the average approximation ratio achieved in 10 independent runs (stated as trend lines in Section 6).

According to Figure 3, for some instances, the average approximation ratios are close to 100%, while the same achievement seems to be very difficult on others. For example, HSGP regularly achieves better results than CS2SA and S5 on almost all instances, and for some other instances from the *kroA100* group, HSGP is better than MA2B.

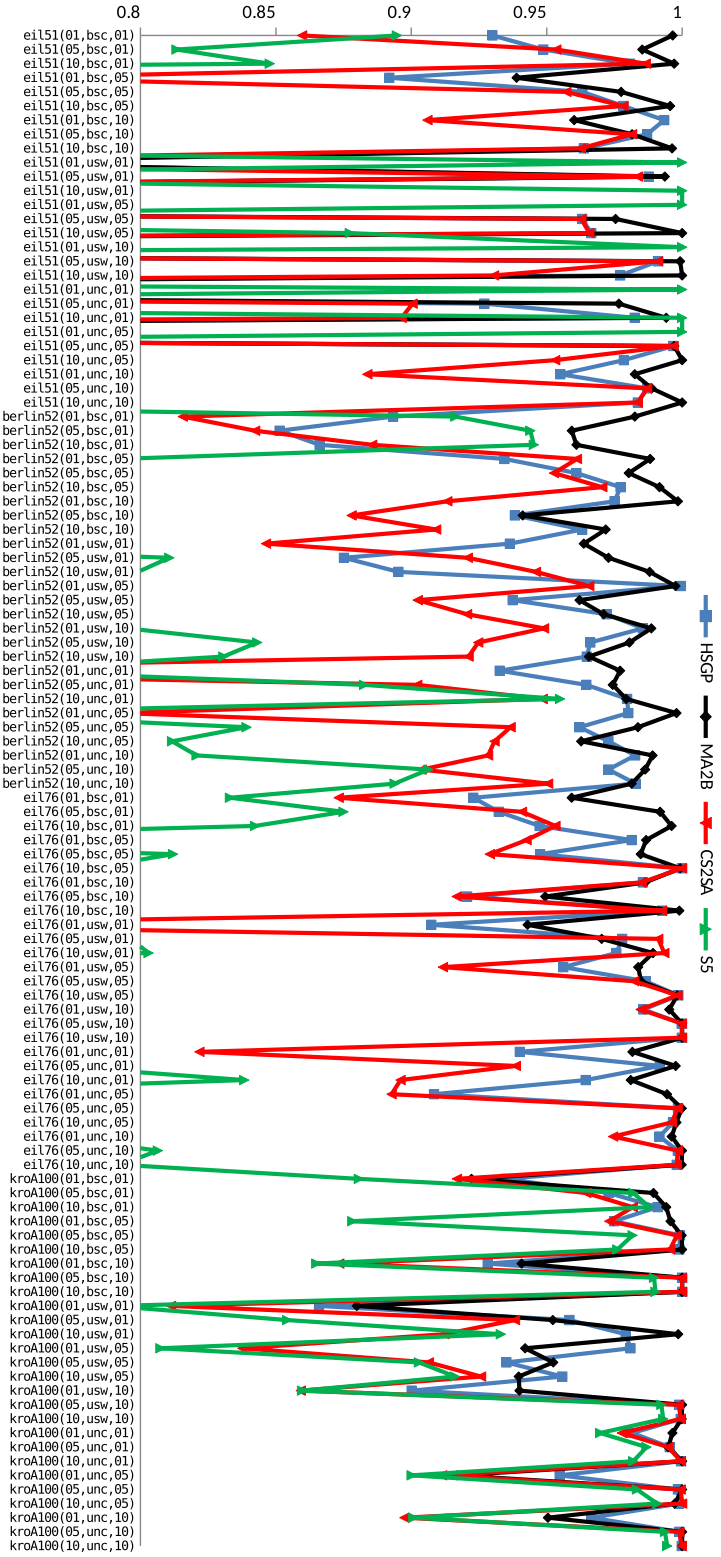

Fig. 3: Average approximation ratios over 10 independent runs.
We use the following naming convention: $instance\_group(F, T, C)$.

References

1. Andreas Bölte and Ulrich Wilhelm Thonemann. Optimizing simulated annealing schedules with genetic programming. *European Journal of Operational Research*, 92(2):402–416, 1996.
2. M. R. Bonyadi, Z. Michalewicz, and L. Barone. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In *Proceedings of the 2013 IEEE Congress on Evolutionary Computation*, pages 1037–1044, June 2013.
3. Mohammad Reza Bonyadi, Zbigniew Michalewicz, Michal Roman Przybylek, and Adam Wierzbicki. Socially inspired algorithms for the travelling thief problem. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, GECCO'14*, pages 421–428. ACM, 2014.
4. Edmund K. Burke, Matthew R. Hyde, Graham Kendall, and John Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO'07*, volume 2, pages 1559–1565, London, 2007. ACM.
5. Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. *Exploring Hyper-heuristic Methodologies with Genetic Programming*, pages 177–201. Springer Berlin Heidelberg, 2009.
6. Edmund K. Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R. Woodward. *A Classification of Hyper-heuristic Approaches*, pages 449–468. Springer US, Boston, MA, 2010.
7. E.K. Burke, G. Kendall, and E. Soubeiga. A Tabu-Search Hyperheuristic for Timetabling and Rostering. *Journal of Heuristics*, 9(6):451–470, 2003.
8. Mauro Castelli, Luca Manzoni, Leonardo Vanneschi, Sara Silva, and Aleš Popovič. Self-tuning geometric semantic genetic programming. *Genetic Programming and Evolvable Machines*, 17(1):55–74, 2016.
9. Peter Cowling, Graham Kendall, and Eric Soubeiga. A Hyperheuristic Approach to Scheduling a Sales Summit. In *Proceedings of the Third International Conference on Practice and Theory of Automated Timetabling, PATAT 2000*, pages 176–190, Konstanz, Germany, 2000. Springer Berlin Heidelberg.
10. Peter Cowling, Graham Kendall, and Eric Soubeiga. A Parameter-Free Hyperheuristic for Scheduling a Sales Summit. In *Proceedings of the 4th Metaheuristic International Conference, MIC 2001*, pages 127–131, 2001.
11. Alberto Cuesta-Cañada, Leonardo Garrido, and Hugo Terashima-Marín. Building Hyper-heuristics Through Ant Colony Optimization for the 2D Bin Packing Problem. In *Proceedings of the 9th International Conference, KES 2005*, pages 654–660, Melbourne, Australia, 2005. Springer Berlin Heidelberg.
12. Kathryn A. Dowsland, Eric Soubeiga, and Edmund Burke. A simulated annealing based hyperheuristic for determining shipper sizes for storage and transportation. *European Journal of Operational Research*, 179(3):759 – 774, 2007.
13. John H Drake, Matthew Hyde, Khaled Ibrahim, and Ender Ozcan. A genetic programming hyper-heuristic for the multidimensional knapsack problem. *Kybernetes*, 43(9/10): 1500–1511, 2014.
14. Mohamed El Yafrani and Belaïd Ahiod. Cosolver2B: An efficient Local Search Heuristic for the Travelling Thief Problem. In *Proceedings of the 2015 IEEE/ACS 12th International Conference of Computer Systems and Applications, AICCSA*, pages 1–5, Nov 2015. doi: 10.1109/AICCSA.2015.7507099.
15. Mohamed El Yafrani and Belaïd Ahiod. A local Search based Approach for Solving the Travelling Thief Problem. *Applied Soft Computing*, 2016. ISSN 1568-4946. doi: http://dx.doi.org/10.1016/j.asoc.2016.09.047.
16. Mohamed El Yafrani and Belaïd Ahiod. Population-based vs. Single-solution Heuristics for the Travelling Thief Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO'16, pages 317–324, Denver, Colorado, USA, 2016. ACM. ISBN 978-1-4503-4206-3.
17. Hayden Faulkner, Sergey Polyakovskiy, Tom Schultz, and Markus Wagner. Approximate approaches to the traveling thief problem. In *Proceedings of the 2015 Annual Conference*

on Genetic and Evolutionary Computation, GECCO'15, pages 385–392. ACM, 2015.
18. Alex S. Fukunaga. Automated Discovery of Local Search Heuristics for Satisfiability Testing. *Evolutionary Computation*, 16(1):31–61, March 2008. ISSN 1063-6560.
19. Rachel Hunt, Kourosh Neshatian, and Mengjie Zhang. A Genetic Programming Approach to Hyper-Heuristic Feature Selection. In *Proceedings of the 9th International Conference on Simulated Evolution and Learning, SEAL 2012*, pages 320–330, Hanoi, Vietnam, 2012. Springer Berlin Heidelberg.
20. Miha Kovačič. Modeling of total decarburization of spring steel with genetic programming. *Materials and Manufacturing Processes*, 30(4):434–443, 2015.
21. Miha Kovačič and Franjo Dolenc. Prediction of the natural gas consumption in chemical processing facilities with genetic programming. *Genetic Programming and Evolvable Machines*, 17(3):231–249, 2016.
22. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.
23. John R. Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, Norwell, MA, USA, 2003. ISBN 1402074468.
24. Natalio Krasnogor and Jim Smith. Emergence of Profitable Search Strategies Based on a Simple Inheritance Mechanism. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, pages 432–439, San Francisco, California, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-774-9.
25. William B. Langdon, Riccardo Poli, Nicholas F. McPhee, and John R. Koza. *Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications*, pages 927–1028. Springer Berlin Heidelberg, 2008.
26. Sean Luke and Liviu Panait. Lexicographic Parsimony Pressure. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO'02, pages 829–836, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. ISBN 1-55860-878-8.
27. Y. Mei, X. Li, F. Salim, and X. Yao. Heuristic evolution with genetic programming for traveling thief problem. In *Proceedings of the 2015 IEEE Congress on Evolutionary Computation, CEC*, pages 2753–2760, May 2015.
28. Yi Mei, Xiaodong Li, and Xin Yao. Improving efficiency of heuristics for the large scale traveling thief problem. In *Proceedings of the Asia-Pacific Conference on Simulated Evolution and Learning*, pages 631–643. Springer, 2014.
29. Yi Mei, Xiaodong Li, Flora Salim, and Xin Yao. Heuristic evolution with genetic programming for traveling thief problem. In *Proceedings of the 2015 IEEE Congress on Evolutionary Computation, CEC*, pages 2753–2760. IEEE, 2015.
30. Yi Mei, Xiaodong Li, and Xin Yao. On investigation of interdependence between subproblems of the travelling thief problem. *Soft Computing*, 20(1):157–172, 2016.
31. Su Nguyen, Mengjie Zhang, and Mark Johnston. A Genetic Programming Based Hyper-heuristic Approach for Combinatorial Optimisation. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO'11, pages 1299–1306, Dublin, Ireland, 2011. ACM. ISBN 978-1-4503-0557-0.
32. Sergey Polyakovskiy, Mohammad Reza Bonyadi, Markus Wagner, Zbigniew Michalewicz, and Frank Neumann. A Comprehensive Benchmark Set and Heuristics for the Traveling Thief Problem. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO'14, pages 477–484, Vancouver, BC, Canada, 2014. ACM. ISBN 978-1-4503-2662-9.
33. Gerhard Reinelt. Tspliba traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.
34. Peter Ross. *Hyper-Heuristics*, pages 529–556. Springer US, Boston, MA, 2005.
35. Peter Ross, Javier G. Marín-Blázquez, Sonia Schulenburg, and Emma Hart. Learning a Procedure That Can Solve Hard Bin-Packing Problems: A New GA-Based Approach to Hyper-heuristics. In *Proceedings of the Genetic and Evolutionary Computation 2003, GECCO'03*, pages 1295–1306, Chicago, IL, USA, 2003. Springer Berlin Heidelberg.
36. Sara Silva. Gplaba genetic programming toolbox for matlab, version 4.0 (2015). University of Coimbra, 2009. URL `http://gplab.sourceforge.net/download.html`.
37. Sara Silva and Jonas Almeida. Gplab-a genetic programming toolbox for matlab. In *Proceedings of the Nordic MATLAB Conference (NMC-2003*, pages 273–278, 2005.

38. Alejandro Sosa-Ascencio, Gabriela Ochoa, Hugo Terashima-Marin, and Santiago Enrique Conant-Pablos. Grammar-based generation of variable-selection heuristics for constraint satisfaction problems. *Genetic Programming and Evolvable Machines*, 17(2): 119–144, 2016.
39. András Vargha and Harold D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.
40. Markus Wagner. Stealing Items More Efficiently with Ants: A Swarm Intelligence Approach to the Travelling Thief Problem. In *Proceedings of the 10th International Conference on Swarm Intelligence, ANTS 2016*, pages 273–281, Brussels, Belgium, 2016. Springer International Publishing.
41. Markus Wagner, Marius Lindauer, Mustafa Misir, Samadhi Nallaperuma, and Frank Hutter. A case study of algorithm selection for the traveling thief problem. *arXiv preprint arXiv:1609.00462*, 2016.
42. R. R. Wilcox. *Statistics for the social sciences*. Academic Press, San Diego, New York, 1996.
43. Junhua Wu, Sergey Polyakovskiy, and Frank Neumann. On the impact of the renting rate for the unconstrained nonlinear knapsack problem. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, GECCO'16*, pages 413–419. ACM, 2016.